NAME :	**Ms Neerja Doshi**
UID :	**2021300029**
DIV :	**SE CE Div A**
DAA Expt 8 : Using branch and bound strategy	

AIM :	To solve the 15-Puzzle Problem using Branch and Bound strategy.
ALGORITHM :	1) The 15-puzzle problem is a classic sliding puzzle game where you have to arrange 2) 15 numbered tiles in a 4x4 grid, where only one tile is empty, and the goal is to reach the final state in which the tiles are arranged in ascending order from left to right, top to bottom. 3) Branch and bound is an algorithmic technique used to solve optimization problems, and it can be applied to solve the 15-puzzle problem. 4) Rabin Karp Algorithm: ◦ It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. 5) **Algorithm :** I. Start. II. Take a node and branch it based on all the possible 1-step moves. III. For each branched node, calculate the cost required to solve the problem. IV. Repeat steps 2 and 3 for the branched node with minimum cost till solved state is reached. V. Prune the other branched nodes. VI. End. 6) **Time Complexity :** ◦ $O(n^2)$

CODE :	//header files
	#include<stdio.h>
	#include<stdlib.h>
	#include<stdbool.h>
	//sample solution(to be used as reference for error count)
	int soln[][4]={
	{1,2,3,4},

```
{5,6,7,8},
{9,10,11,12},
{13,14,15,0}
};
//node structure
struct Node{
int errorCount;
int blankTileLoc[2];
int puzzle[4][4];
char currMove;
struct Node* parent;
struct Node** children;
};
//global variables
struct Node* root;
int finalCost;
bool allowedMoves[4];//order taken into consideration- R, L, U, D
char movesOrder[4]={'R', 'L', 'U', 'D'};
//function for swapping values at two locations
void swap(int* a, int* b){
int temp=*a;
*a=*b;
*b=temp;
}
//functions for getting the change factor to move the blank tile
int changeInX(int i){
if(i==0||i==1)
return 0;
else if(i==2)
return -1;
else
return 1;
}
int changeInY(int i){
if(i==2||i==3)
return 0;
else if(i==1)
return -1;
elsereturn 1;
}
//function to determine the number of possible moves
int movesCount(struct Node* node){
int count=4;
if(node->blankTileLoc[1]+1==4||node->currMove=='L'){
count--;
allowedMoves[0]=false;
}
if(node->blankTileLoc[0]+1==4||node->currMove=='U'){
```

```
count--;
allowedMoves[3]=false;
}
if(node->blankTileLoc[1]-1==-1||node->currMove=='R'){
count--;
allowedMoves[1]=false;
}
if(node->blankTileLoc[0]-1==-1||node->currMove=='D'){
count--;
allowedMoves[2]=false;
}
return count;
}
//function to copy the puzzle
void copyPuzzle(int arr1[][4], int arr2[][4]){
for(int i=0; i<4; i++){
for(int j=0; j<4; j++)
arr1[i][j]=arr2[i][j];
}
}
//function to print the puzzle structure
void printPuzzle(struct Node* node){
for(int i=0; i<4; i++){
for(int j=0; j<4; j++)
printf("%d\t",node->puzzle[i][j]);
printf("\n");
}
}
//solving the puzzle using bound and branch strategy
struct Node* solvePuzzle(struct Node* node, int cost){
if(node->errorCount==0){
finalCost=cost;
return node;
}
//initially considering that all moves are allowed
for(int i=0; i<4; i++)
allowedMoves[i]=true;
int possibleMoves=movesCount(node);
//branching the node for exploration
node->children=(struct Node**)malloc(possibleMoves*sizeof(struct
Node*));for(int i=0; i<possibleMoves; i++){
node->children[i]=(struct Node*)malloc(sizeof(struct Node));
copyPuzzle(node->children[i]->puzzle,node->puzzle);
node->children[i]->parent=node;
node->children[i]->errorCount=node->errorCount;
}
//exploration
int minErr=16, indexOfMinErr=0, childIndex=-1;
```

```
int xCor, yCor;
for(int i=0; i<4; i++){
if(allowedMoves[i]){
childIndex++;
xCor=node->blankTileLoc[0];
yCor=node->blankTileLoc[1];
swap(&node->children[childIndex]->puzzle[xCor][yCor],&node-
>children[childIndex]->puzzle[xCor+changeInX(i)]
[yCor+changeInY(i)]);
node->children[childIndex]->currMove=movesOrder[i];
node->children[childIndex]->blankTileLoc[0]=xCor+changeInX(i);
node->children[childIndex]->blankTileLoc[1]=yCor+changeInY(i);
if(node->children[childIndex]->puzzle[xCor][yCor]==soln[xCor]
[yCor])
node->children[childIndex]->errorCount-=1;
else
node->children[childIndex]->errorCount+=1;
if(node->children[childIndex]->errorCount<minErr){
minErr=node->children[childIndex]->errorCount;
indexOfMinErr=childIndex;
}
}
}
printf("\nMove-%d: %c\n\n",cost+1,node->children[indexOfMinErr]-
>currMove);
printPuzzle(node->children[indexOfMinErr]);
//branching the min-cost child node while bounding others
return solvePuzzle(node->children[indexOfMinErr],cost+1);
}
void printSolution(struct Node* node){
if(node->parent->parent!=NULL){
printSolution(node->parent);
printf("->%c",node->currMove);
}
else{
printf("%c",node->currMove);
}
}
//main function
void main(){
//initializing the root node
root=(struct Node*)malloc(sizeof(struct Node));
root->parent=NULL;
root->currMove='N';//N means no move done yet
//taking the inputs for root nodeprintf("\nEnter the initial state of
the puzzle(0 for empty tile)-\n\n");
for(int i=0; i<4; i++){
for(int j=0; j<4; j++)
```

```
scanf("%d",&(root->puzzle[i][j]));
}
//checking the number of errors in the initial state
root->errorCount=0;
for(int i=0; i<4; i++){
for(int j=0; j<4; j++){
if(root->puzzle[i][j]==0){
root->blankTileLoc[0]=i;
root->blankTileLoc[1]=j;
}
if(root->puzzle[i][j]!=soln[i][j]&&root->puzzle[i][j]!=0)
root->errorCount++;
}
}
printf("\nFollowing is the step-by-step solution of the puzzle-\n");
struct Node* solnNode=solvePuzzle(root,0);
printf("\nSummary-");
printf("\n\nTotal Number of moves required to solve the puzzle
----->
%d\n",finalCost);
printf("Following is the step-by-step movement of blank tile to
solve the puzzle: ");
printSolution(solnNode);
printf("\n\n");
}
```

OUTPUT :

```
Enter the initial state of the puzzle(0 for empty tile)-

1       2       3       4
5       6       0       8
9       10      7       11
13      14      15      12

Following is the step-by-step solution of the puzzle-

Move-1: D

1       2       3       4
5       6       7       8
9       10      0       11
13      14      15      12

Move-2: R

1       2       3       4
5       6       7       8
9       10      11      0
13      14      15      12

Move-3: D

1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      0

Summary-

Total Number of moves required to solve the puzzle -----> 3
Following is the step-by-step movement of blank tile to solve the puzzle: D->R->D
```

CONCLUSION :	By Performing the above experiment , Ive understood branch and biund strategy to solve 15 puzzle problem.